Interactive scalable condensation of reverse engineered UML class diagrams for software comprehension

Osman, M.H.B.

Chapter 2

Definitions

In this chapter, we define common terms used in this research. We start with describing the term 'software comprehension' because it is a common theme throughout this thesis. We also describe 'cognitive design elements for software exploration' because they are the basis for representing the results of our approach in a graphical visualization. Also, 'forward-' and 'reverse engineering' process are explained because we use the information in these processes as the primary input in our studies. We briefly describe UML because this design notation is the central notation used in this research. We explain the basics of machine learning because this method is used to classify the importance of elements of the software design.

2.1 Software Comprehension

Software comprehension (also known as program-/code-/system- comprehension) is a critical activity in software maintenance. Software comprehension is a complicated activity which requires a lot of time, effort and cost. In particular, studies have shown that more than 50% of the time spent in software maintenance is spent on trying to understand source code [20][23]. Software developers must familiarize themselves when they are new to a system, but it is also the case that they must familiarize themselves with new parts of a system as the system evolves. Table 2.1 lists several definitions on program comprehension as found in the literature. Based on these definitions, we define program comprehension in this research as:

Definition 2.1. "Program comprehension is a process of understanding a program based on available software artifacts (such as documentations, as well as source code)."

Biggerstaff (1993) [25] characterizes what aspects of the software must be understood through the comprehension process: *"A person understands a program when they*

Table 2.1: *Definitions of Program Comprehension*

Author(s)	Definitions
Rugaber (1995) [88]	*"Program comprehension as the process of acquiring knowledge about a computer program"*
Cimitile (2000) [89]	*"Program comprehension is the process of acquiring sufficient knowledge about a software artifact so as to be able to successfully accomplish a given task"*
Ng et al. (2004) [116]	*"Program comprehension is the process of understanding a program through feature and documentation analysis"*
Maalej et al. (2014) [107]	*"Program comprehension is the activity of understanding how a software system or a part of it works"*

are able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program."

Program comprehension is a tedious challenge that requires a lot of effort. Hence, we believe that facilitating program comprehension will help software developers in this activity. Several models have been proposed that characterize the cognitive process of program comprehension. We discuss these program comprehension models in the next subsection.

2.1.1 Program Comprehension Model

A software developer's approach to program comprehension may be bottom-up, top-down or a combination of both. This is relevant in the context of this thesis as it led us to develop methods to view class diagrams at different levels of abstraction. We briefly discuss these program comprehension models below.

Bottom-up Comprehension

Bottom-up comprehension models denote the bottom-up building of system understanding: by reading the code and then rationally chunking or grouping these statements or representation into higher-level abstractions. These chunks (or abstractions) are aggregated further until the entire system is comprehended [152].

The Pennington comprehension model [135] describes two program abstractions formed by software developers during comprehension [43]: (1) A *program model* is a low-level abstraction consisting of knowledge of operations at a level close to the surface of the program code and of control flow relations indicating the flow of execution, and (2) A *domain model* (or *situation model*) is a higher-level abstraction comprising knowledge of data flow and functional relationships. This comprehension model is frequently

chosen when the source code or domain of the system is not familiar to the software developer.

Top-down comprehension

The top-down comprehension model (based on Brooks's model [37]) denotes a hypothesis-driven comprehension approach. This approach begins with the software developers making a general hypothesis about the program's function. Such hypothesis are formed based on information outside the program code such as design documents or system descriptions. This initial hypothesis leads software developers to anticipate to certain structures (objects and operations) in the program, producing another level of more distinct hypotheses (sub-hypotheses). At this point, the software developer has concrete things to look for in the program code, for which hypothesis verification is attempted [43].

In contrast to the bottom-up approach, the top-down comprehension is usually used for a system and a domain that is familiar to the software developer.

Integrated Model

Von Mayrhauser (1993) [173] introduced an integrated model that combines the top-down and bottom-up model. This model is motivated by the argument that software developers may switch flexibly between top-down to bottom-up comprehension models depending on the situation. In addition to the top-down and bottom-up comprehension model, the integrated model also involves the knowledge base (typically known as human mind) that stores (1) any new information acquired straightforwardly from the system; or (2) indirect information. Figure 2.1 illustrates this model and more detail on the terms and terminology of this figure can be found in [174].

2.1.2 Cognitive Design Elements for Software Exploration

The time consumed for understanding existing implementation code is a significant proportion of the time needed for maintaining, debugging and reusing the existing code. This explains the importance of tool support for software comprehension. Tools in this area can be characterized as being a software visualization tool or a software exploration tool. Schäfer et al. [150] indicated that software visualization tools (e.g., [100],[86],[49]) assemble visualization techniques to intensify understanding, while software exploration tools (e.g. [57],[145]) offer an essential software navigation facility (searching/browsing/summarizing/condensing). However, there is no concrete boundary between these categories. In this research, we focus on software exploration tools as we aim to provide an automated tool to explore the software design based on the reverse engineered class diagram.

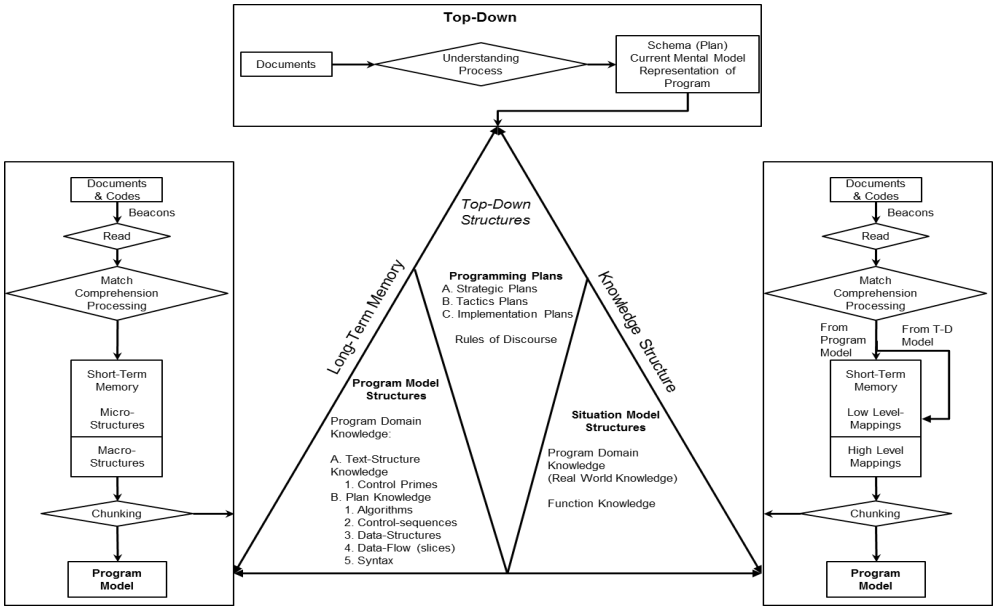

Figure 2.1: *Integrated Model [174]*

A common feature in software exploration tools is the graphical representation of the system structure together with the corresponding source code. Storey et al.[157] introduced a set of guidelines for software exploration tools (as shown in Figure 2.2). These guidelines are called '*cognitive design elements*'. From the guidelines that she suggests, we prioritize a number which we aim to fulfill in our approach. The tool developed in this research focuses on comprehension of software design. Hence, we would like to incorporate the following guidelines to our tool: **E3**: Provide an abstraction mechanism; **E4**: Support goal-directed hypothesis-driven comprehension; **E5**: Provide an adequate overview of the system architecture at various levels of abstraction; **E6**: Support the construction of multiple *mental models*[1] (domain, situation, program); **E11**: Indicate the maintainer's current focus; and **E15**: Provide effective presentation style.

2.2 Forward and Reverse Engineering

In software engineering, the software development life cycle (SDLC) is a schema that characterizes the process of developing software. In other words, it is a conceptual model used in software project management that describes the stages involved in system development from preliminary feasibility review through maintenance. The

[1] A *mental model* describes developer's mental representation of the system-to-analyze

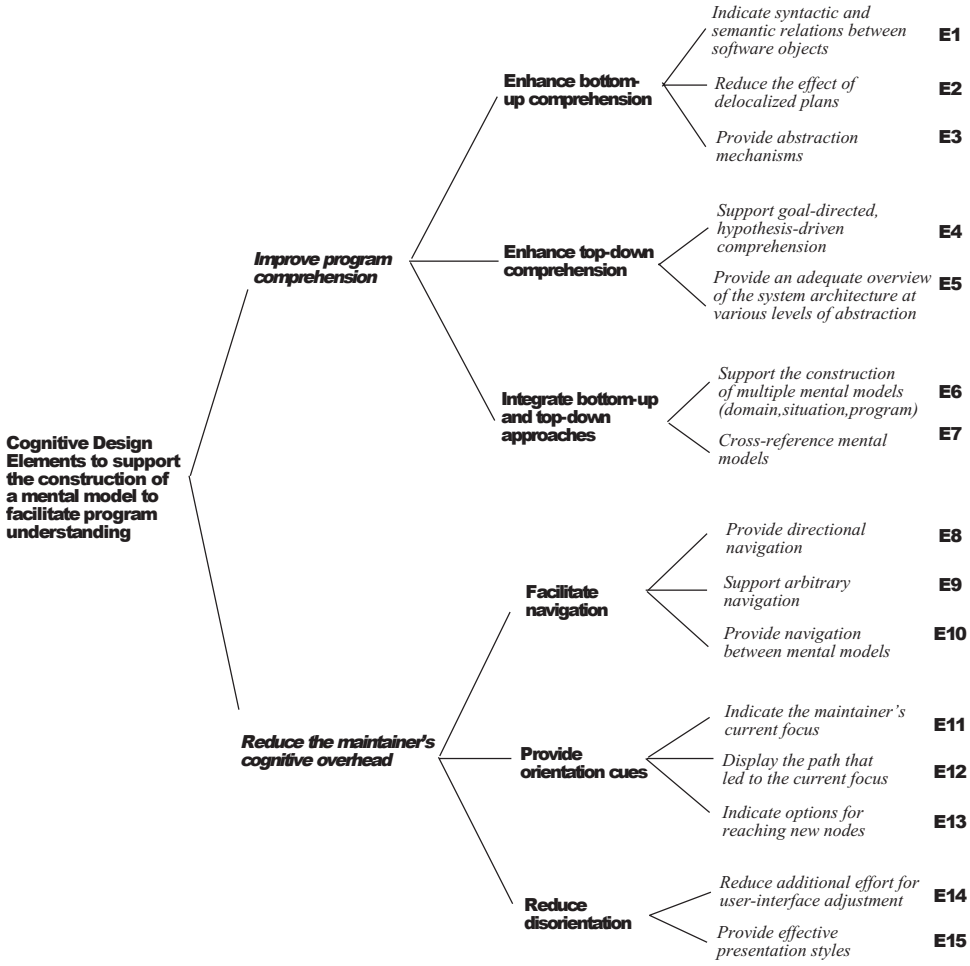

Figure 2.2: *Cognitive Design Elements for Software Exploration [156]*

SDLC describes software development stages (at a high level) as: analysis, design, implementation and maintenance. In this research, we focus on three stages of the SDLC as illustrated in Figure 2.3 (stages that are coloured with blue). We use artifacts from the design and implementation stage (as our input) to produce a result (output) for the usage in the maintenance stage. The term software maintenance (from SDLC) is typically used in this research. The following definition of software maintenance by IEEE std 1219-1998 [1] is used in this research:

Definition 2.2. "Software maintenance is a modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."

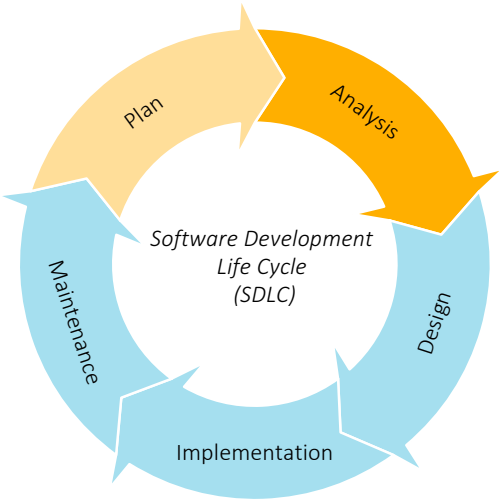

Figure 2.3: *The Software Development Life Cycle*

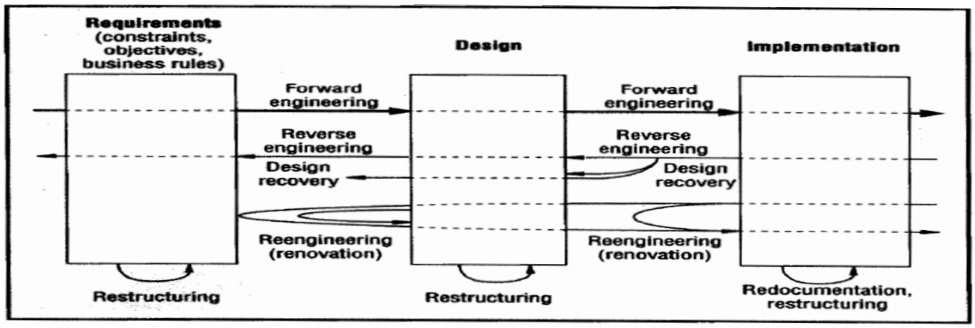

Figure 2.4: *Relationship between Forward Eng., Reverse Eng. and Other Related Terms [41]*

Regarding these stages (see Figure 2.4), there are two processes related to this research: Forward Engineering and Reverse Engineering. The following subsections briefly explain these processes.

2.2.1 Forward Engineering

The following definition of forward engineering by Chikofsky and Cross [41] is used in this research.

Definition 2.3. *"Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system."*

The term forward design (FD) is used to indicate a design produced in a forward engineering routine (constructed after the requirement stage). The FD usually addresses the functional requirement of the system (mostly specific to the system domain) and also the non-functional requirements.

2.2.2 Reverse Engineering

The following definition of reverse engineering by Chikofsky and Cross (1990) [41] is used in this research.

Definition 2.4. *"Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships, and create representations of the system in another form or at a higher level abstraction."*

In general context, reverse engineering is the process of recovering knowledge of software, based on the existing software artifacts. There are different types of techniques can be used in the reverse engineering process, most importantly: static- and dynamic analysis. We describe these next.

2.2.3 Static and Dynamic Analysis

This subsection briefly describes the definition of static and dynamic analysis that we use in this research. For static analysis, the following definition by Jarzabek (2007) [85] is used in this research.

Definition 2.5. *"Static analysis means an analysis of a program text, without executing a program, such as is typically done by a compiler front end."*

In most cases, the analysis is performed on the source code, and in other cases, it is performed on the object code and execution file. Static analysis is suitable for recovering the system structure (e.g. structural design).

For dynamic analysis, the following definition by Bell (1999) [19] is used in our research:

Definition 2.6. *"Dynamic analysis is the analysis of the properties of a running program. Dynamic analysis derives properties that hold for one or more executions by examination of the running program."*

An advantage of using dynamic analysis is its ability to detect objects dependencies (at runtime). However, dynamic analysis cannot guarantee to discover the complete functionality of the system for an overview for the developer. Amongst others, this is because dynamic analysis can only pass through a limited subset of functions in the systems within any practical time-bound. In addition, the source code (and hence modules) that are passed through typically depend on the set of input values, which for most practical purposes has infinitely many combinations. In this thesis, we work

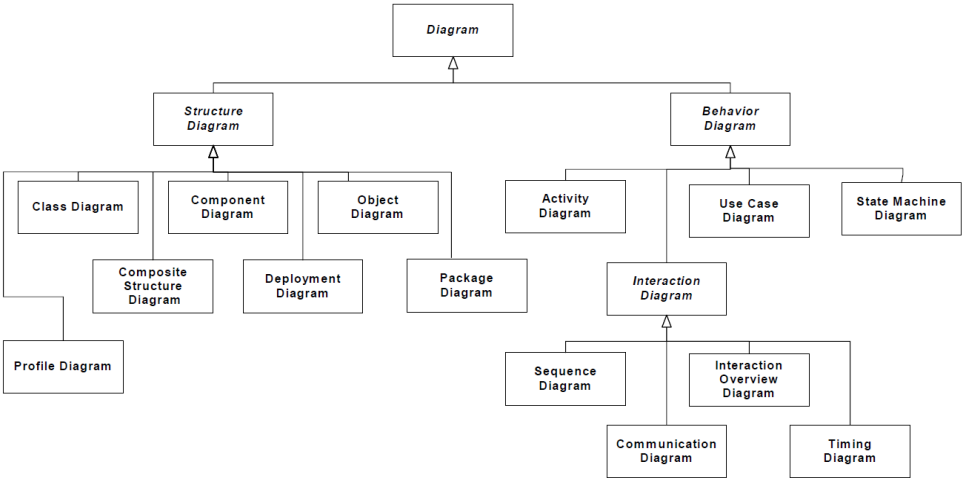

Figure 2.5: *Taxonomy of UML Version 2.4 [68]*

purely with static analysis for the recovery of the structure of software designs from source codes.

Next, we explain the UML notation which is used for representing this static design structure of software.

2.3 The Unified Modeling Language

The Unified Modeling Language (UML) is a graphical notation intended to provide a standardized communication tool in software development. In practice, software is almost always developed by a group of software engineers. For this reason, a communication tool is critical to ensure good communication because the software design should be well understood by everyone working on the project.

UML was introduced in 1997, and was developed based on existing object-oriented design methods, namely the object-modeling technique (OMT) [146], Booch [31], and Object-Oriented Software Engineering (OOSE) [83]. There are two kinds of UML Diagrams: structure- and behavior diagrams. A taxonomy of UML diagrams is shown in Figure 2.5. The structure diagrams demonstrate the structural parts of the system at diverse levels of abstraction as well as how structural parts are related to one another. On the other hand, the behavior diagrams demonstrate the behavior of a system, which could be portrayed as an arrangement of actions of the system over time. Through supporting both these structure and behavior diagrams, UML provides a graphical representation of a system during design, implementation, as well as the maintenance phase. The next subsection describes the UML-related terms that are commonly used in this research.

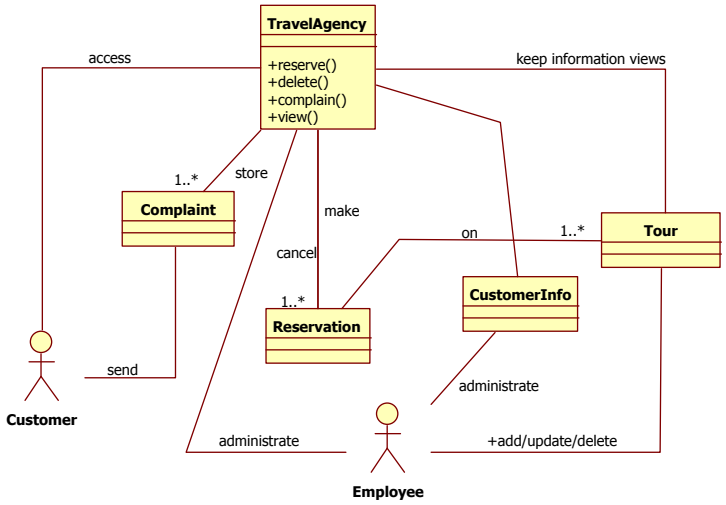

Figure 2.6: *Tours Online Class Diagram (Domain Analysis)*

2.3.1 UML Class Diagram

The most common diagram to demonstrate the structural view of a system is the class diagram [147]. This view illustrates a collection of classes, possibly interfaces, and relations between classes. Relationships that hold between classes convey critical information about the design. The basic types of relationships in UML are association (including aggregation and composition) and generalization (also known as inheritance). These relations provide the foundation of the system structure.

Class diagrams can be used throughout the software development life cycle (SDLC), due to the fact that this diagram may carry diverse types of information - depending on the SDLC processes and on the level of detail being considered. At the beginning of the SDLC, the class diagram may be used to reflect the software requirements (*domain analysis class diagram*). As development progresses, class diagrams can be used to represent information that is more relevant to the construction of the system (*design level class diagram*[2]). During or after the implementation of source code, a class diagram may be recovered using reverse engineering techniques. Such a reverse engineered class diagram is closely based on the source code and reflect the fine-grain implementation structure of software systems. We call such reverse engineered class diagrams as RE-CD. Figure 2.6, 2.7 and 2.8 illustrate these different types of class diagrams. These examples of class diagrams are taken from Jalloul [84].

[2]Depending on the level of detail (LoD) of the class diagram, *design level class diagrams* may be turned to *code level class diagrams* if a high LoD is applied.

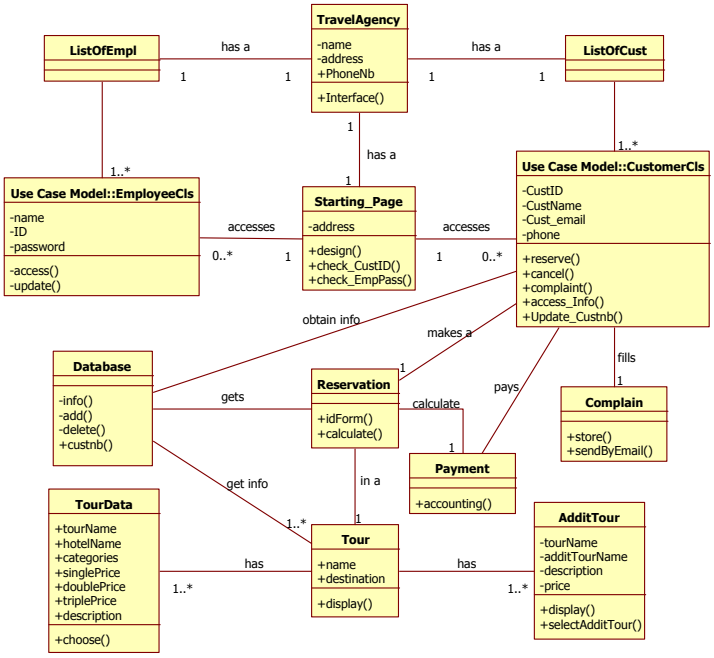

Figure 2.7: *Tours Online Class Diagram (Design Level)*

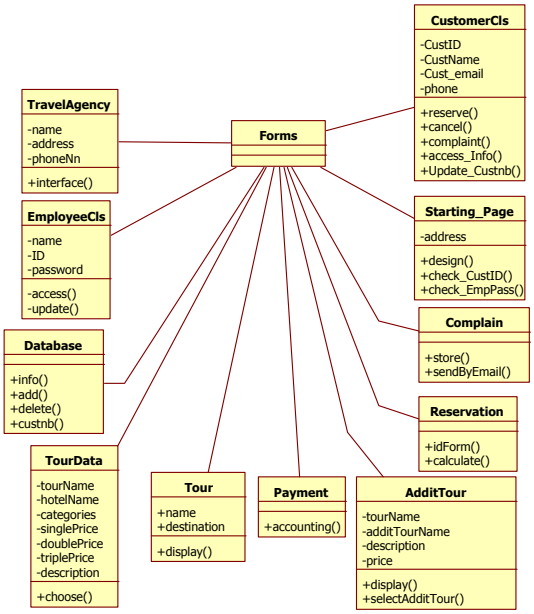

Figure 2.8: *Tours Online RE-CD (Code Level)*

2.3.2 UML Class Diagram for Software Comprehension

A graphical representation (e.g. UML class diagram) can help software engineers to comprehend large-scale systems. However, their effectiveness is subject to the syntax and semantics of UML, spatial diagram layout and domain knowledge [166]. Yusuf et al. [183] show that experts (in class diagrams) tend to use such things as stereotype[3] information, colouring and layout to facilitate more efficient exploration and navigations of class diagrams.

In this research, we focus on RE-CDs that are close to source code (*code level class diagrams*). The stereotype information is not available in RE-CD and hence this information is not used in our approach.

The work by [160], [183], [71], [151], [158] and [159] demonstrate the effect of layout on system comprehension. However, we believe that the choice of layout is subjective and highly depends on the user expertise and purpose of using the diagram. Thus, we did not cover the layout of the RE-CD in this thesis. The aspect of layout remains open or for future research. Nonetheless, in our research, we apply a colouring technique to highlight those classes that are important in the class diagram. This colouring technique aims to help the software developers to focus on the important classes.

2.3.3 XML Metadata Interchange

XMI stands for Extensible Markup Language (XML [176]) Metadata Interchange. It is a standard for representing UML models using XML. The current version released by the OMG is XMI 2.4.1 which has been formally published by the International Organization for Standardization (ISO) as ISO/IEC 19509:2014 Information technology – Object Management Group XML Metadata Interchange (XMI)[69].

The objective of XMI is to allow simple interchange of metadata between UML modeling tools and Meta Object Facility (MOF)-based repositories within distributed heterogeneous environments. The standards that are related to XMI are the following:

1. **UML** – Unified Modeling Language (ISO/IEC 19505)
2. **MOF** – Meta Object Facility (ISO/IEC 19508)
3. **XML** – eXtensible Markup Language [176]

The MOF defines a standard metamodel for applications, allowing UML models to be interchanged among tools and repositories; XMI standardizes the format for these interchanges [66]. It utilizes XML schemas to describe object-oriented models and enable interoperability between UML-based tools. XMI is flexible. Thus, the XML representation can be tailored to suit the user requirements. Most of the UML tools extend the XMI format with their proprietary information, which result in that other UML-based tools can not completely and correctly read the XMI file. This issue has

[3] A stereotype is a special type of class that represents a domain-specific concept. Graphically, a stereo-type class can be adorned with a special graphical form or decorations so that it stands out from generic classes.

been raised by Stevens [155] in 2003, but it still exists after more than ten years. In this research, we aim to parse as much XMI flavours and versions as possible; to provide an automated tool that usable for many XMI formats and hence UML tools. Finding a way to solve this issue is one of the practical challenges in this research.

2.4 Machine Learning

This research aims at building a method to decide: what classes could be included and what classes could be excluded in class diagrams in order to facilitate system comprehension. Two approaches can be used to build this method:

1. *Rule-based Approach*
   A rule-based system consists of if-then rules, facts, and an interpreter controlling the application of the rules. Conventional rule-based expert systems, use human expert knowledge to solve real-world problems that normally would require human intelligence [12] and,
2. *Machine Learning Approach*
   The machine learning approach is useful for domains where humans might not have the knowledge needed to develop effective algorithms, where the program must dynamically adapt to changing conditions [113].

Based on the datasets (see Chapter 3), we discovered that the machine learning approach is more suitable to be used for our research. The reason is that there is no explicit knowledge available in the dataset. Furthermore, the dataset is coming from different types of domain and sizes. It is difficult for such context-sensitive problem to be solved by using a rule-based approach. Also, the following reasons motivate us to choose machine learning as the approach for class inclusion/exclusion selection:

- It provides algorithms that may facilitate to automatically[4] classify the important classes in a class diagram based on the training data (in our case, training data are gathered from forward design and RE-CD);
- Human resources are not required to formulate rules. Therefore, it may avoid the inefficiency of human learning [16].
- It considers context and it can utilize multiple sources of knowledge to formulate the classification rules.
- It can adapt if new information becomes available.

Next, we explain the definitions of machine learning, types of machine learning, machine learning classification algorithms and performance measure for classification algorithms.

[4] With no or little human intervention.

2.4.1 Definition of Machine Learning

The common definitions of machine learning are the following:
Arthur Samuel (1959) [149][113] defines machine learning as:

Definition. *" A field of study that gives computers the ability to learn without being explicitly programmed."*

This definition outlines the basic concept of machine learning. Later, Mitchel (1997) [113] introduces a further formalized definition of machine learning as shown in the following.

Definition. *"Well-posed Learning Problem: A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."*

However, Witten et al. (2005) [178] more focus on the 'descriptions' model (from examples). They define machine learning as:

Definition. *"The acquisition of structural descriptions from examples. The kind of descriptions found can be used for prediction, explanation, and understanding."*

There are several types of machine learning provided to solve problems (depends on the purpose and available data). The types of machine learning are described in the next section.

2.4.2 Types of Machine Learning

The following describes four major types of machine learning algorithms:

- *Supervised machine learning* is the search for algorithms (see Figure 2.9) that reason from externally supplied instances to produce general hypotheses, which then make predictions about future instances [98]. The training data (input) have a known label or predefined result, for example, a binary label of buy/not buy in the stock market. Through the training process, a model is constructed to make predictions of test instances (data). Examples of supervised learning tasks are classification and regression.
- *Unsupervised machine learning* learns to characterize certain input pattern in a fashion that reflects the statistical structure of the overall collection of input patterns [45]. The input data are unlabeled, and no predefined results are provided. The goal of unsupervised learning is to find some kind of structure in the data. An example of a common unsupervised learning task is clustering.
- *Semi-supervised learning* is halfway between unsupervised and supervised learning [38]. The input data is a mixture of unlabeled and labeled sample. The purpose of semi-supervised learning is to discover how combining labeled with unlabeled data may change the learning behavior, and design algorithms that benefit from such a combination.

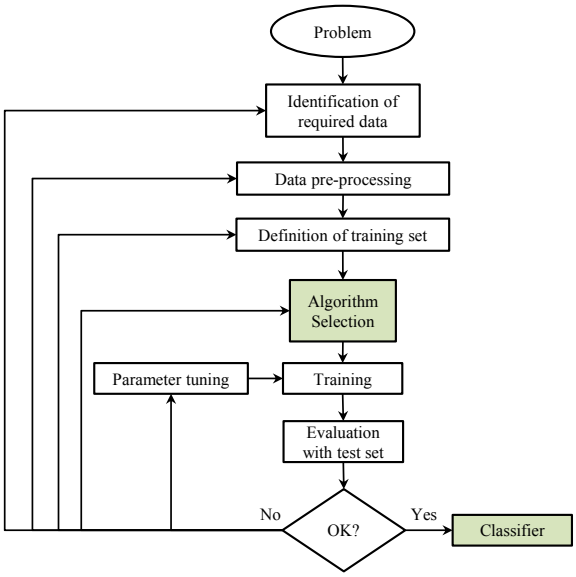

Figure 2.9: *The Process of Supervised Machine Learning [98]*

- *Reinforcement machine learning* is the learning of a mapping from situations to actions so as to maximize a scalar reward or reinforcement signal [161]. The learner is not told which action to take, but must discover which action results in the best reward by trying them. The action may affect not only the immediate reward, but also the next situation through all subsequent rewards.

This research applies supervised machine learning classification algorithms to decide what classes could be included and what classes could be excluded (omitted) in class diagrams. Next, we explain the supervised machine learning classification algorithms that are used in our experiments.

2.4.3 Machine Learning Classification Algorithms

This subsection focuses on several supervised machine learning classification algorithms that we believe are suitable for the purpose of this research. As illustrated in Figure 2.9, a selection of classification algorithm(s) is needed to make sure our proposed framework uses the algorithm(s) that fit with the datasets and the purpose of research. Prior to making a selection of the classification algorithms, several exploratory experiments on a wider range of algorithms need to be conducted. In this research, we do not expect that there will be a single silver bullet algorithm that will outperform all others across all sets of problems. Also, we are not just interested in a single algorithm that scores a top result on a given problem, but are looking for sets of

classifiers[5] (i.e. classification algorithms) that produce robust results across domains. In this way, algorithms become more portable across problems with very different rates of inclusion of classes in designs. We also aimed for a mix of classifiers in terms of expected bias (what relationships can be captured) and variance (does the prediction change when trained on different random samples) [171].

As discussed, we want to use a diverse set of algorithms representative for different approaches. For example, Decision Trees, Stumps, Tables and Random Trees or Forests all divide the input space up in disjoint smaller sub-spaces and make a prediction based on the occurrence of positive classes in those sub-spaces. K-Nearest-Neighbour (k-NN) and Radial Basis Functions (RBF) Networks are similar local approaches, but the sub-spaces here are overlapping. In contrast, Logistic Regression and Naive Bayes model parameters are estimated based on potentially large numbers of instances and can thus be seen as more global models. The nine classification algorithms we consider are described in Table 2.2 (refer [178] for more explanation).

Most of the classification algorithms in our experiments are designed to produce a predicted outcome class label[6] for each test instance [58]. However, this research aims to produce ranking of classes on the importance; hence, we are more interested in the *classifier score* for every instance rather than just a set of instance classification labels. A high *classifier score* of a class indicates the class is important while a lower *classifier score* indicates the class is less important.

The classification algorithm(s) for this research are selected based on the classification performance and their robustness to all datasets. We explain the performance measure of classification algorithms in the next section.

2.4.4 Performance Measure For Classification Algorithms

A performance measure of machine learning classification algorithms can be derived from a confusion matrix (as shown in Table 2.3). Several performance measures to compare classification algorithms (formulated based on the confusion matrix) are described in Table 2.4 (refer [178] for more detail). Our datasets used in this research are typically imbalanced (i.e. low proportion of classes in forward design and high proposition of classes in RE-CD, as shown in Chapter 3, 7 and 8). Hence, the common performance measures listed in Table 2.4 do not fit for our purpose. Referring to the confusion matrix example in Table 2.3, the overall success rate (accuracy) is 95.24%. It seems that the algorithm performs an excellent prediction. The 95.24% is calculated by taking the sum of correct prediction divided by the overall number of predictions. The percentage of correct prediction for TN is 98.8%, while TPR is 25%. The resulting prediction performance for TP is very low, even though overall correct prediction is very high.

[5] In this thesis, *classifier* refers to classification algorithms models, not the term classifier in UML.

[6] A binary-classification that attempts to produce 'Yes' or 'No' class labels.

Table 2.2: *The nine classification algorithms*

Algorithms	Description
Decision Table	A Decision Table consists of rows and columns that associate a set of conditions or tests with a set of actions. The machine learning tool used in this research - Waikato Environment for Knowledge Analysis (WEKA) [76] uses a simple Decision Table Majority (DTM) classifier.
Decision Stumps	Decision Stumps are decision trees consisting of just a single level and split [171]. A decision stump makes a prediction based on the value of just a single input feature, and is a good baseline classifier to compare against decision trees and other classifiers, to determine what results can already be achieved with a very basic model.
J48 Decision Tree (J48)	J48 is a WEKA implementation of the C.45 decision tree algorithm [178]. This algorithm generates a classification-decision tree for the given dataset by recursive partitioning of data.
k-Nearest Neighbour (k-NN)	k-NN classification finds a group of k objects in the training set that are most similar to the test object and bases its classification on the predominance of a particular class in this neighborhood [179].
Logistic Regression (LR)	LR uses a linear input combination of input variables to provide an output score, which is then mapped to a probability by applying a logistic function [61].
Naive Bayes (NB)	NB is a classification algorithm based on the Bayes rule of conditional probability. It assumes that the presence/absence of a particular feature of a class is unrelated (independence) to the presence/absence of any other feature [110].
Radial Basis Func. (RBF) Networks	RBF Networks are a type of feed-forward neural network. We used simple normalized Gaussian functions that each cover part of the input space and the activation of each of these functions given an output is then fed into a basic feed-forward neural network [120].
Random Forests	Random Forests is a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution of all trees in the forest [33].
Random Tree	The Random Tree algorithm builds a classification algorithm tree considering K randomly chosen predictors at each node. More explanation of Random Tree is provided in [101].

Table 2.3: *Confusion Matrix or Contingency Table*

Prediction Result		**Actual Result**
Y	**N**	
TP	*FN*	**Y**
FP	*TN*	**N**
Example:		
Y	**N**	
11	*33*	**Y**
10	*849*	**N**

Note :

True Positive (**TP**) : A positive instance that is correctly classified as positive
False Positive (**FP**) : A negative instance that is incorrectly classified as positive
True Negative (**TN**) : A negative instance that is correctly classified as negative
False Negative (**FN**) : A positive instance that is correctly classified as negative

Table 2.4: *Common Performance Measures and Terms*

Terms and Measures	**Description**
Overall Success Rate or Accuracy (*Acc*) [15]	$Acc = \frac{TP + TN}{TP + TN + FP + FN}$
True Positives Rate (*TPR*) or Recall or Sensitivity [58]	$TPR = \frac{TP}{TP + FN}$
False Positives Rate (*FPR*) [58] or false alarm ratio	$FPR = \frac{FP}{FP + TN}$
Precision	$Precision = \frac{TP}{TP + FP}$
F-measure (F_1) [26]	$F_1 = 2 * (\frac{Precision * Recall}{Precision + Recall})$

In this research, we search for algorithms that provide reliable estimates across the score range, thus we evaluate using the Area Under ROC[7] Curve (AUC) [79] value rather than accuracy. For imbalanced data, the AUC also avoids the issue of favouring models that just predict the majority outcome class. The larger the ROC area, the better the classification algorithm is in term of classifying classes [14]. The AUC value (calculated using WEKA) measures the performance of a model over the entire range of model scores, i.e. how well it separates by changing the score threshold of a class over the entire score range. Therefore, AUC shows the ability of the classification algorithms to rank classes correctly as more likely to be included in the class diagram or not. AUC is quite often be used to evaluate classification algorithms that utilized imbalanced dataset [39].

Precision and recall are common in information retrieval for evaluating classification performance [58]. However, these performance measures are not suitable to be used for our dataset (due to imbalanced or *class skew*). The F-measure aims to improve by balancing precision and recall, but the issue is that it still needs a fixed classification threshold[8] (in our case, there is no specified threshold as we aim to cover the whole range of scores). Therefore, AUC is preferred over accuracy, precision, recall and F-Measure (refer [62] for further discussion). The AUC measure is based on ROC graphs; A two-dimensional graph in which *TPR* is plotted on the Y-axis and *FPR* is plotted on the X-axis (see Figure 2.10 (a)). It indicates relative tradeoffs between true positives and false positives. Figure 2.10 [58] compares two classifiers evaluated using ROC curves and precision-recall curves. Figure 2.10 (a) and (b) show a balanced dataset (1:1 class distribution) while Figure 2.10 (c) and (d) show an imbalance dataset (1:10 class distributions[9] of the same classifier and same domain). This figure demonstrates that the ROC curves (Figure 2.10 (a) and (c)) are identical, but the precision-recall curves (Figure 2.10 (b) and (d)) differ substantially.

2.5 Summary

In this chapter, we defined the key concepts that are used in this research. We described the UML as our focus of this research. In particular, this research utilized the forward design and RE-CD as the main input. We use XMI as the input because it can be generated by most of the common CASE tools. Machine learning is the heart of this research as we use the classification algorithms to classify the classes that could be included and classes could be omitted in order to simplify the class diagram. The list of classified classes (included or excluded) from the classification process is meaningless if

[7] ROC means Receiver Operating Characteristics.

[8] A decision threshold is the cut-off degree employed to decide the final prediction of a classification model. In binary classification, the final prediction is class positive if the model's posterior probability of a test example is above the threshold; or else it is class negative [168].

[9] The classifier and the underlying concept are the same; different only in class distribution

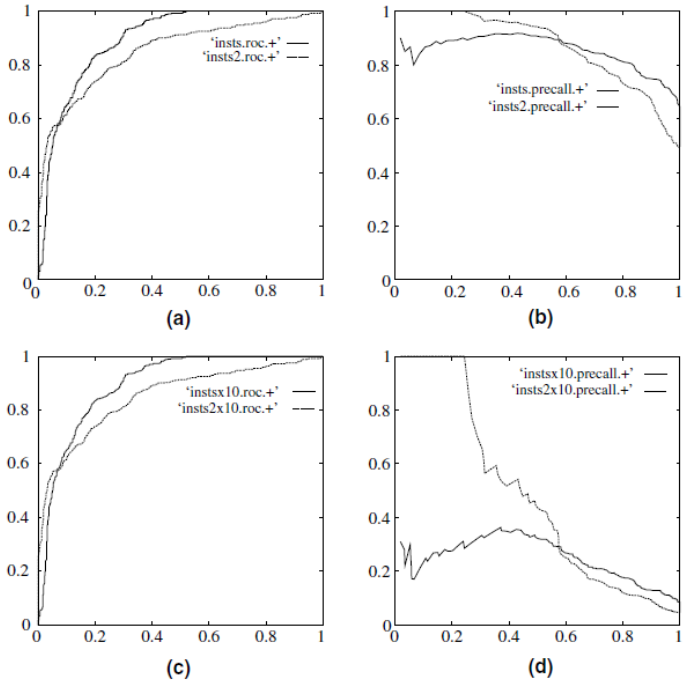

Figure 2.10: *ROC and Precision-Recall Curves under Class Skew*
(a) ROC curves (1:1 distribution) (b) precision-recall curves, (1:1 distribution)
(c) ROC curves, (1:10 distributions) (d) precision-recall curves, (1:10 distributions)

it is not presented graphically. Therefore, we refer to several cognitive design elements for software exploration that we believe useful to assist the software developer in understanding software.

This chapter only defines the common terms of this research. Other related terms and also related works are presented in each chapter.